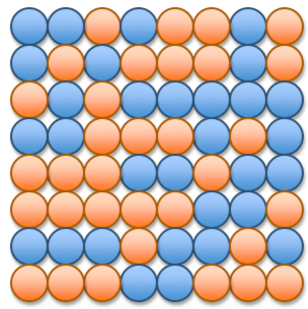


UNIT 7B

Data Representation: Compression

Fixed-Width Encoding

- In a fixed-width encoding scheme, each character is given a binary code with the same number of bits.
  - Example:
    Standard ASCII is a fixed width encoding scheme, where each character is encoded with 7 bits.
    This gives us 2^7 = 128 different codes for characters.

Fixed-Width Encoding

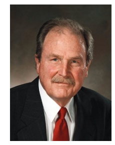
- Given a character set with n characters, what is the minimum number of bits needed for a fixed-width encoding of these characters?
  - Since a fixed width of k bits gives us n unique codes to use for characters, where $n = 2^k$.
  - So given n characters, the number of bits needed is given by $k = \lceil \log_2 n \rceil$. (We use the ceiling function since $\log_2 n$ may not be an integer.)
  - Example: To encode just the alphabet A-Z using a fixed-width encoding, we would need $\lceil \log_2 26 \rceil = 5$ bits: e.g. A => 00000, B => 00001, C => 00010, ..., Z => 11001.

Using Fixed-Width Encoding

- If we have a fixed-width encoding scheme using *n* bits for a character set and we want to transmit or store a file with *m* characters, we would need *mn* bits to store the entire file.
- Can we do better?
  - If we assign fewer bits to more frequent characters, and more bits to less frequent characters, then the overall length of the message might be shorter.

Huffman Coding

- We can use an encoding scheme named after David A. Huffman to compress our text without losing any information.



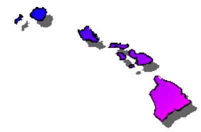
- Based on the idea that some characters occur more frequently than others.

- Huffman codes are not fixed-width.

Huffman Coding: the process

1. Assign character codes
   a. Obtain character frequencies
   b. Use frequencies to build a *Huffman tree*
   c. Use tree to assign variable-length codes to characters (store them in a table)
2. Use table to encode (compress) ASCII source file to variable-length codes
3. Use tree to decode (decompress) to ASCII

The Hawaiian Alphabet



- The Hawaiian alphabet consists of 13 characters.
  - **'** is the okina which sometimes occurs between vowels (e.g. **KAMA'AINA**)
- The table to the right shows each character along with its relative frequency in Hawaiian words.

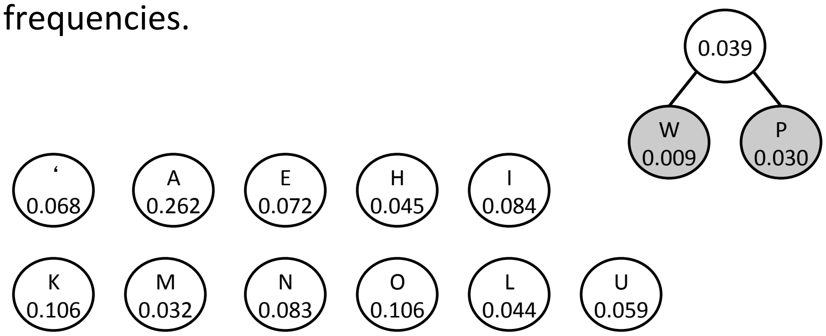
'	0.068
A	0.262
E	0.072
H	0.045
I	0.084
K	0.106
L	0.044
M	0.032
N	0.083
O	0.106
P	0.030
U	0.059
W	0.009

The Huffman Tree

- We use a tree structure to develop the unique binary code for each letter.
- Start with each letter/frequency as its own node:

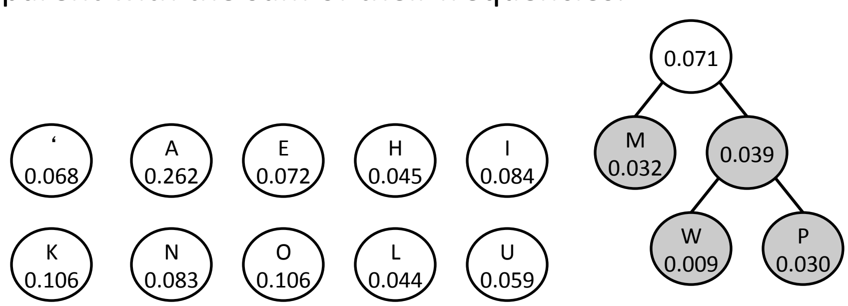
The Huffman Tree

- Combine lowest two frequency nodes into a tree with a new parent with the sum of their frequencies.



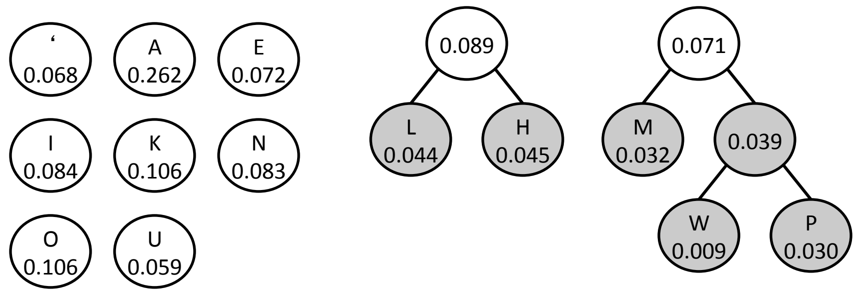
The Huffman Tree

- Combine lowest two frequency nodes (including the new node we just created) into a tree with a new parent with the sum of their frequencies.



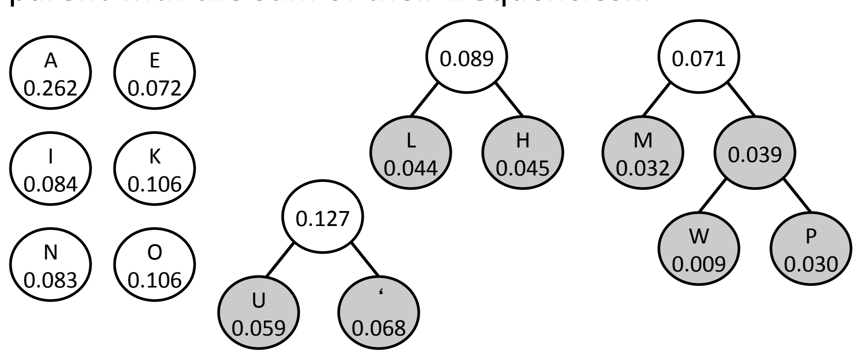
The Huffman Tree

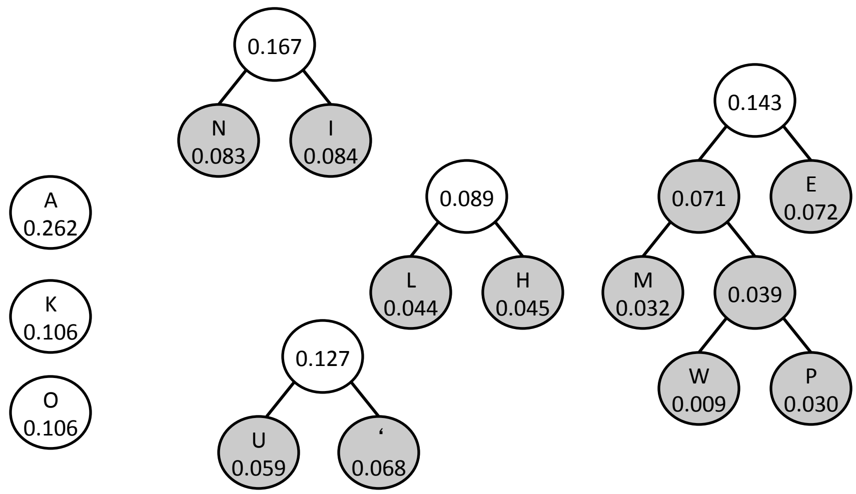
- Combine lowest two frequency nodes (including the new node we just created) into a tree with a new parent with the sum of their frequencies.



The Huffman Tree

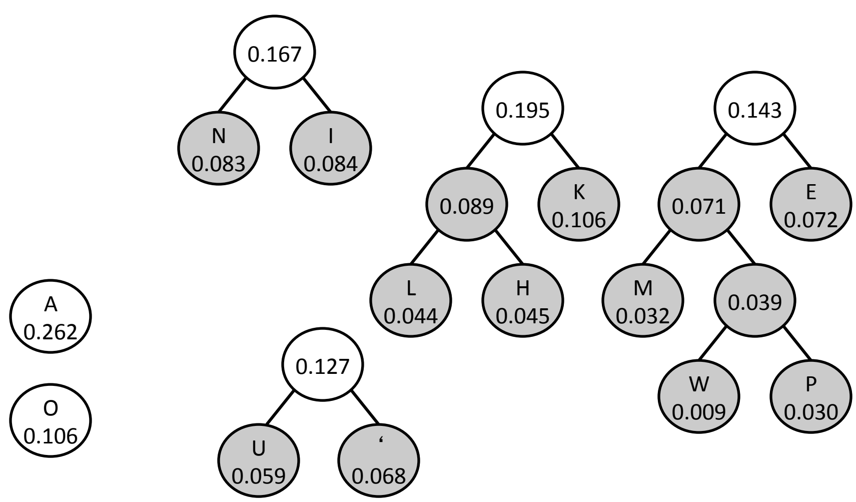
- Combine lowest two frequency nodes (including the new node we just created) into a tree with a new parent with the sum of their frequencies...





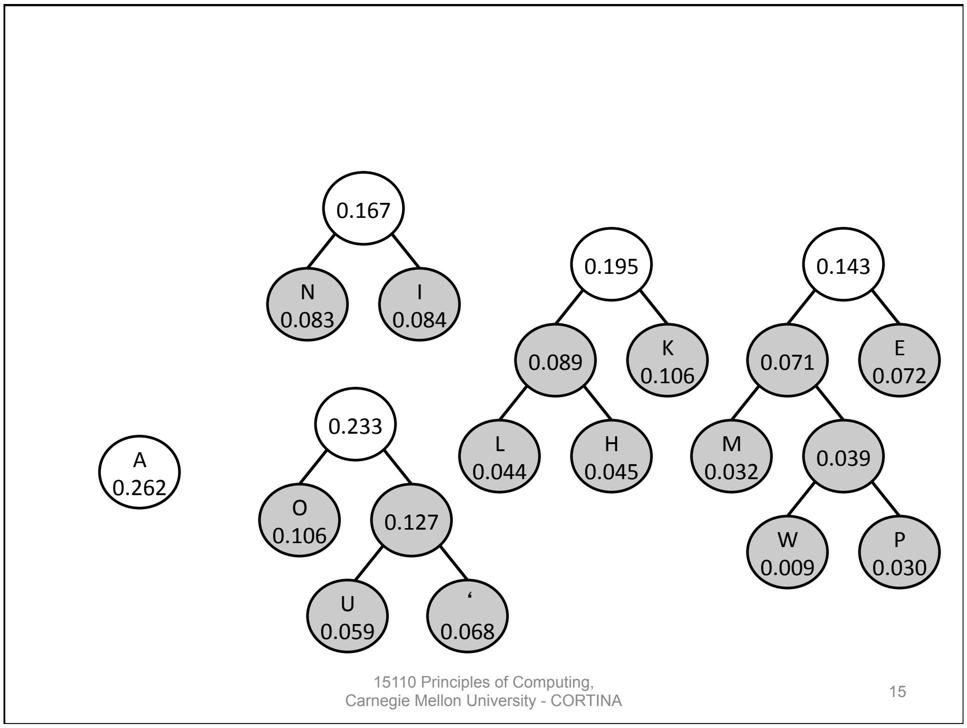
15110 Principles of Computing,
Carnegie Mellon University - CORTINA

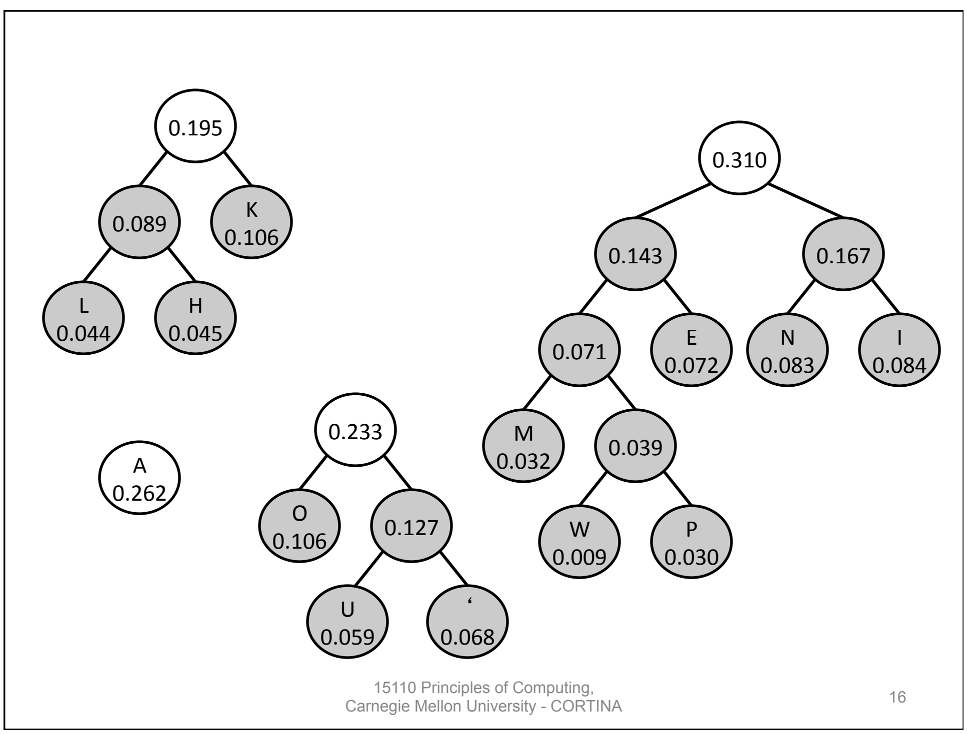
13

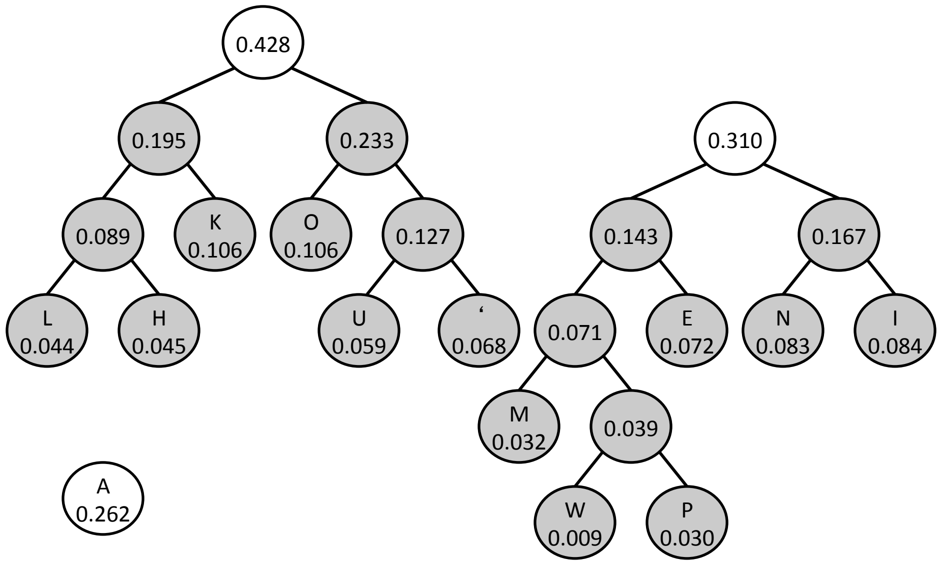


15110 Principles of Computing,
Carnegie Mellon University - CORTINA

14

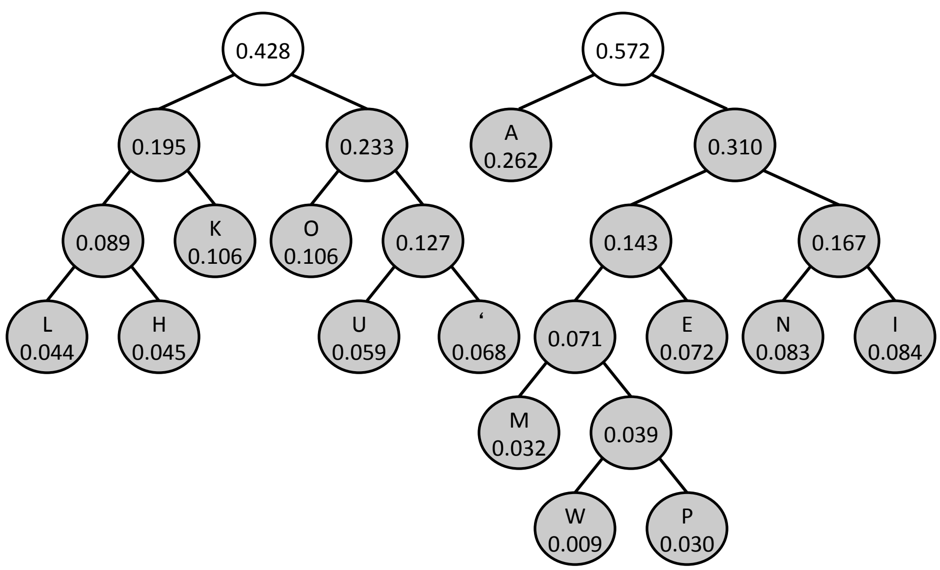

0.167
N 0.083
I 0.084
0.195
0.089
K 0.106
L 0.044
H 0.045
0.143
0.071
E 0.072
M 0.032
0.039
W 0.009
P 0.030
A 0.262
0.233
O 0.106
0.127
U 0.059
‘ 0.068
15110 Principles of Computing,
Carnegie Mellon University - CORTINA
15


0.195
0.089
K 0.106
L 0.044
H 0.045
0.310
0.143
0.167
0.071
E 0.072
N 0.083
I 0.084
M 0.032
0.039
W 0.009
P 0.030
A 0.262
0.233
O 0.106
0.127
U 0.059
‘ 0.068
15110 Principles of Computing,
Carnegie Mellon University - CORTINA
16



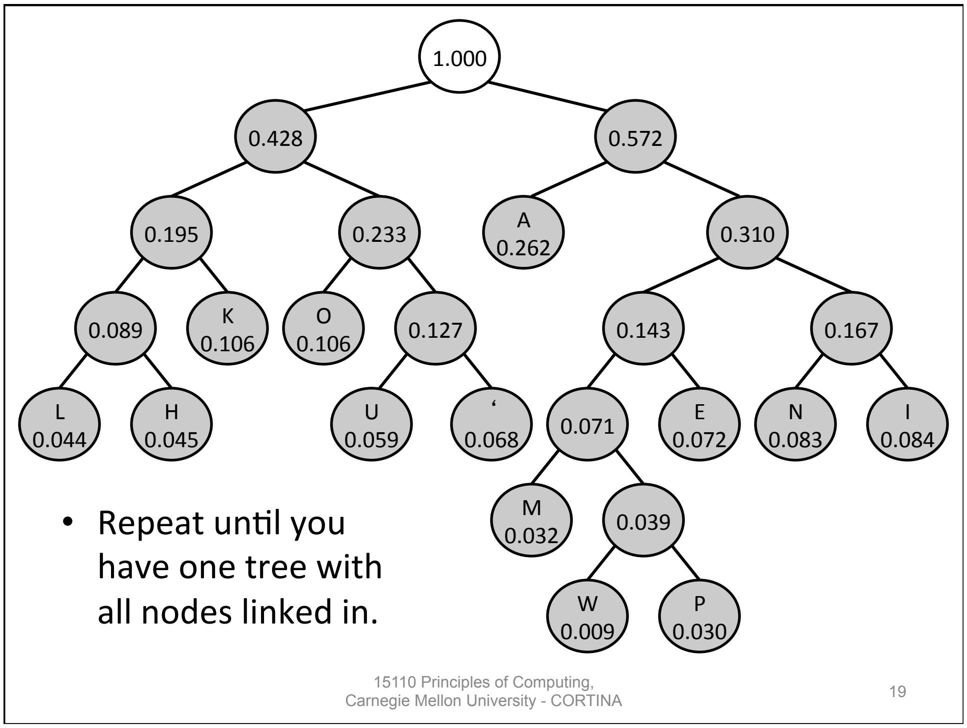
15110 Principles of Computing,
Carnegie Mellon University - CORTINA

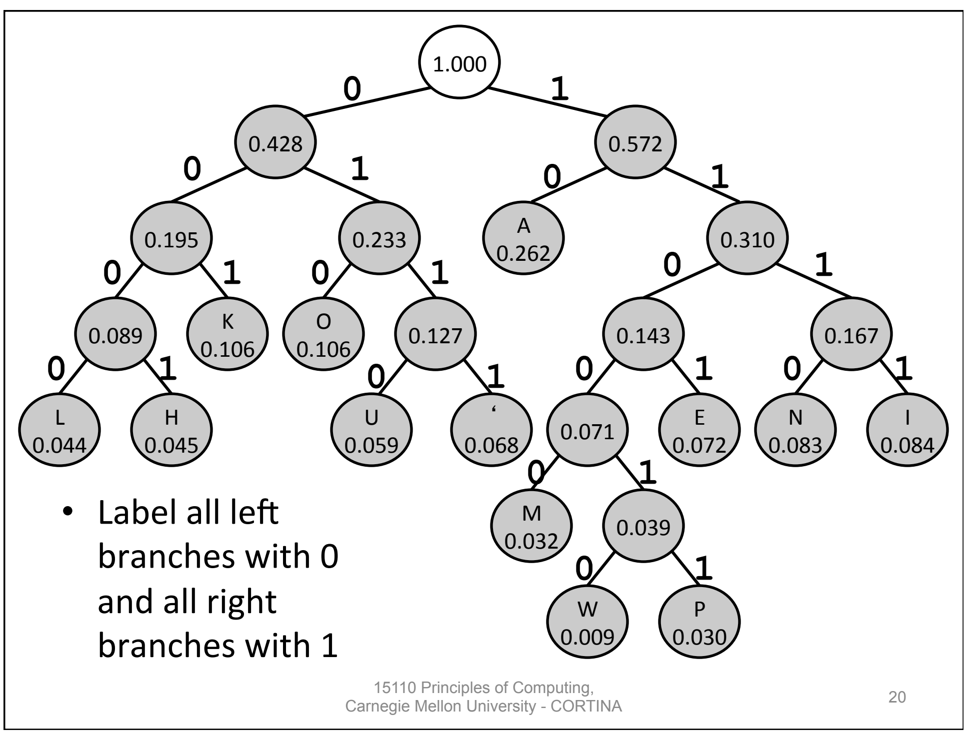
17

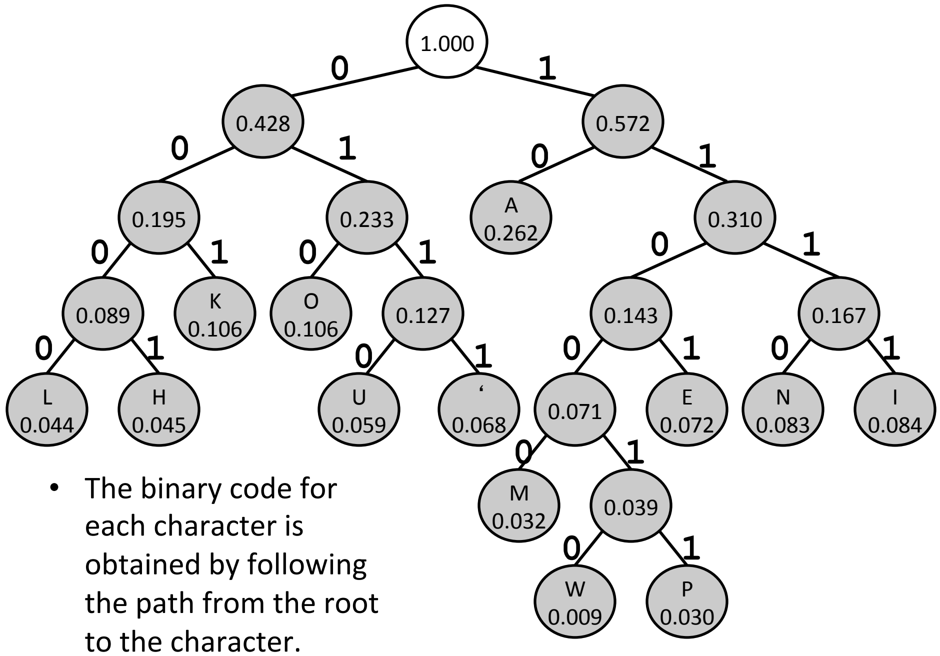


15110 Principles of Computing,
Carnegie Mellon University - CORTINA

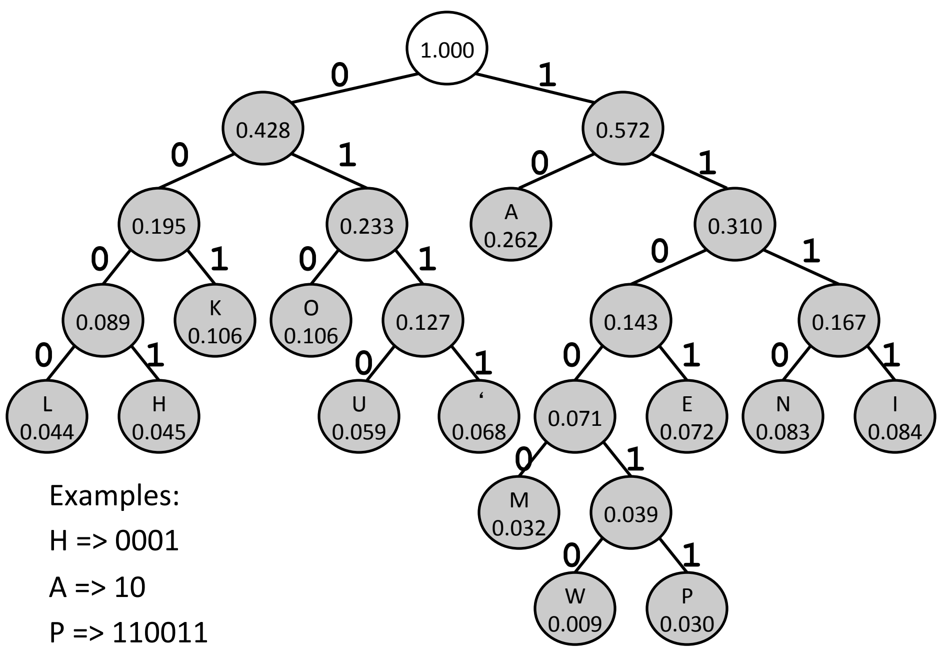
18


• Repeat until you have one tree with all nodes linked in.
15110 Principles of Computing, Carnegie Mellon University - CORTINA
19


• Label all left branches with 0 and all right branches with 1
15110 Principles of Computing, Carnegie Mellon University - CORTINA
20



- The binary code for each character is obtained by following the path from the root to the character.



Examples:

H => 0001

A => 10

P => 110011

Fixed Width vs. Huffman Coding

'	0000	'	0111
A	0001	A	10
E	0010	E	1101
H	0011	H	0001
I	0100	I	1111
K	0101	K	001
L	0110	L	0000
M	0111	M	11000
N	1000	N	1110
O	1001	O	010
P	1010	P	110011
U	1011	U	0110
W	1100	W	110010

ALOHA

Fixed Width:
00010110100100110001
20 bits

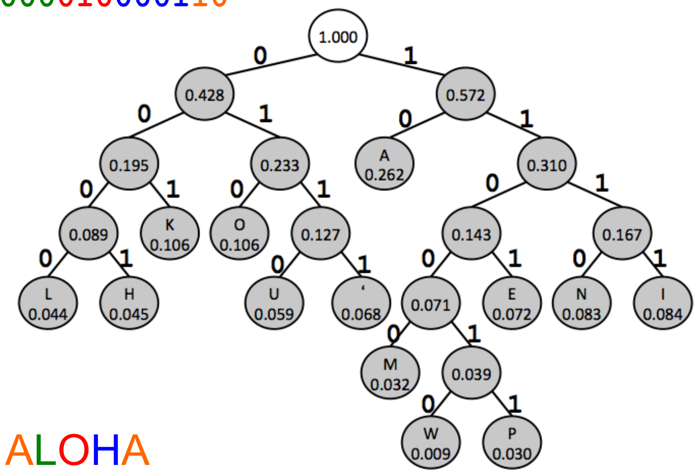
Huffman Code:
100000010000110
15 bits

Decoding

- In a fixed-width code, the boundaries between letters are fixed in advance:
  0001 0110 1001 0011 0001
- With a variable-length code, the boundaries are determined by the letters themselves.
  - No letter's code can be a prefix of another letter.
  - Example: since A is "10", no other letter's code can begin with "10". All the remaining codes begin with "00", "01", or "11".

Decoding

100000010000110



ALOHA

- To find the character use the bits to determine path from root

Programming the Huffman Tree

- Let's write Python code to produce a Huffman tree for a given alphabet.
- At each step we need to find the two nodes with the lowest frequency scores.
- This will be easy if nodes are kept in a list that is sorted by score value.
- Solution: use a **priority queue**.

Priority Queues

NOTE: For this unit, we use PythonLabs (on the linux server) and we need to include the following line in the code:

```
from PythonLabs.BitLab import PriorityQueue, Node,
assign_codes, encode, decode
```


Priority Queue: a data structure

- A priority queue (PQ) is like a list that is automatically kept sorted.
  ```
  >>> pq = PriorityQueue()
  >>> pq
  []
  ```
- PQ methods: **`insert`** and **`pop`**

Priority Queue: insert

- To add an element into the priority queue in its correct position, we use the **`insert`** method:

```
>>> pq.insert("peach”)
>>> pq.insert("apple”)
>>> pq.insert("banana”)
>>> pq
[apple, banana, peach]
```

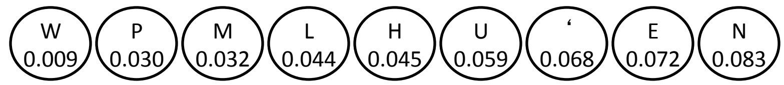

Priority Queue: pop

- To get the first (highest priority) element of the queue, use the **pop** method, which removes it as well:

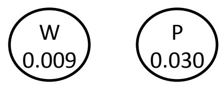
```
>>> fruit1 = pq.pop()
>>> fruit1
'apple'
>>> pq
[banana, peach]
>>> fruit2 = pq.pop()
>>> fruit2
banana
>>> pq
[peach]
```

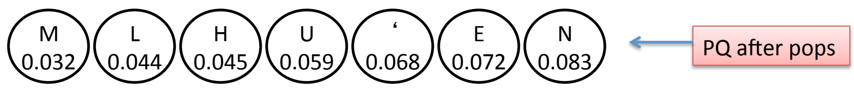
Using a PQ to build the tree

- Make a PQ of `Nodes`. Frequency = priority



To get the two lowest frequency nodes, pop twice





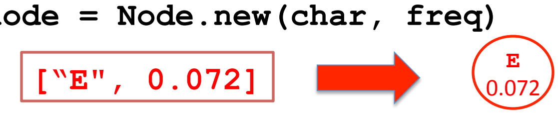
Making Tree Nodes

- Store the character and frequency data into a nested list:

```
table = [ ["'", 0.068], ["A", 0.262],
  ["E", 0.072], ["H", 0.045], ["I", 0.084],
  ["K", 0.106], ["L", 0.044], ["M", 0.032],
  ["N", 0.083], ["O", 0.106], ["P", 0.030],
  ["U", 0.059], ["W", 0.009] ]
```

- Making one of the tree nodes:

```
char = table[2][0]         # "E"
freq = table[2][1]         # 0.072
node = Node.new(char, freq)
```



Building a PQ of Single Nodes

```
def make_pq(table):
    pq = PriorityQueue()
    for item in table:
        char = item[0]
        freq = item[1]
        node = Node(char, freq)
        pq.insert(node)
    return pq
```

Remember: each item in the table is a 2-element list with a character and a frequency.

Building our Priority Queue

```
>>> pq = make_pq(table)
pq
    [( W: 0.009 ), ( P: 0.030 ),
     ( M: 0.032 ), ( L: 0.044 ),
     ( H: 0.045 ), ( U: 0.059 ),
     ( ': 0.068 ), ( E: 0.072 ),
     ( N: 0.083 ), ( I: 0.084 ),
     ( K: 0.106 ), ( O: 0.106 ),
     ( A: 0.262 )]
```

One tree node

Priority queue showing the 13 nodes in sorted order based on frequency.

Building a Huffman Tree

```
def build_tree(pq):
    while len(pq) > 1:
        node1 = pq.pop()
        node2 = pq.pop()
        pq.insert(Node(node1, node2))
    return pq[0]
```

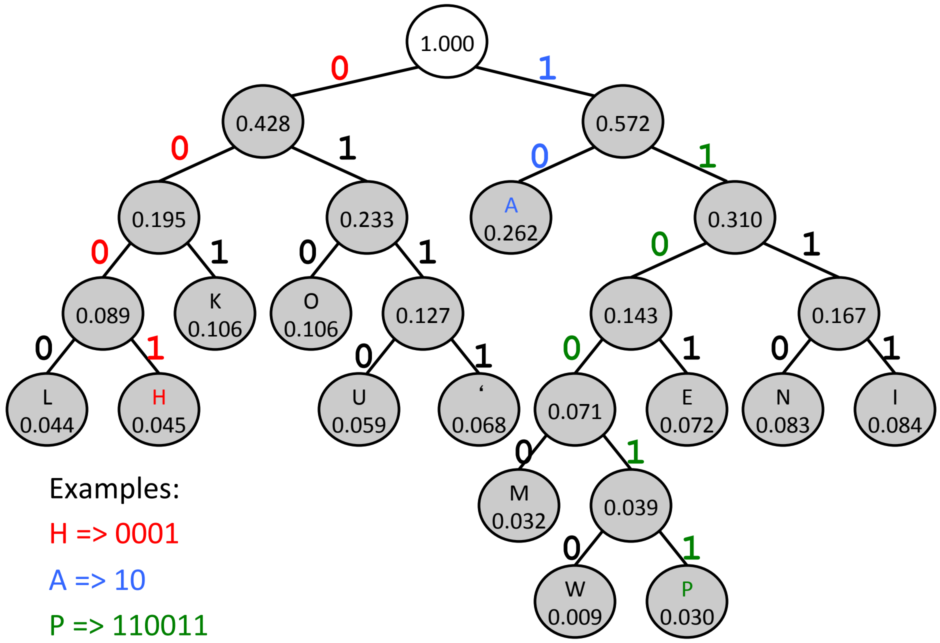
Creates a new node with node1 as its left child and node2 as its right child

(Unlike book version we already created the pq)

Building our Huffman Tree

This is just our Huffman tree expressed using recursively nested parenthetical components:
`( root ( left )`
`( right ) )`

```
tree = build_tree(pq)
⇒ ( 1.000
   ( 0.428
    ( 0.195
     ( 0.089 ( L: 0.044 ) ( H: 0.045
     ( K: 0.106 ) )
    ( 0.233
     ( O: 0.106 )
     ( 0.127 ( U: 0.059 ) ( ': 0.068 ) ) ) )
   ( 0.572
    ( A: 0.262 )
    ( 0.310
     ( 0.143
      ( 0.071 ( M: 0.032 )
       ( 0.039 ( W: 0.009 ) ( P: 0.030 ) ) )
       ( E: 0.072 ) )
      ( 0.167 ( N: 0.083 ) ( I: 0.084 ) ) ) ) )
```



Examples:

H => 0001

A => 10

P => 110011

Assigning Codes, Encoding & Decoding

```
>>> ht = assign_codes(tree)

>>> ht["W"]
110010
>>> ht["A"]
10

>>> msg = encode("ALOHA", tree)
100000010000110
>>> decode(msg, tree)
"ALOHA"
```

from BitLab
takes a Huffman tree and returns a hash table that maps each letter to its binary code

Note the [ ] syntax.
This returns the code associated with the character from the hash table.

from BitLab
encode and decode functions